Paper 100-2009

Customizing DI Studio with a Plug-in to Turn off AutoMap

Chris Olinger, d-Wise Technologies, Inc., Raleigh, NC
Stephen Baker, d-Wise Technologies, Inc., Raleigh, NC

ABSTRACT

SAS® Data Integration Studio (DI Studio) is a desktop client for interfacing with the SAS® Metadata Server for creating, managing, and running processes (SAS® programs) that leverage metadata in the server. This paper explores the available API for extending DI Studio by building a custom plug-in to perform a task that is not available in the delivered out-of-the-box feature set. One of the contentious features available in DI Studio is AutoMap, an option that causes some developers to cringe. An example plug-in is described that leverages the DI Studio API to provide a single click interface to turn off AutoMap on all nodes of a job. Lessons learned and best practices uncovered in the process of building this plug-in are shared.

INTRODUCTION

DI studio comes off the shelf with a number of features designed to save developers time. By leveraging metadata in the SAS® Metadata Server, DI Studio provides users with the ability to quickly build jobs that would otherwise be tedious to create and difficult to visualize. By providing a drag-and-drop GUI to build jobs, DI Studio also allows users to build complex jobs without requiring them to understand too much about writing code. Beyond simply developing jobs, DI Studio provides an interface to the jobs stored in the SAS® Metadata Server and becomes a one-stop tool for managing programs. These features are a powerful combination providing ETL developers a collaborative environment to share code, build jobs, and manage processes. An in-depth discussion of the DI Studio features is not the intention of this paper; however, we will explore two features: 1) the plug-in API, and 2) AutoMap.

DI Studio provides out of the box support for building custom plug-ins to enhance the core feature set of the product and provide additional capabilities in the form of a plug-in API. Some of the features that are delivered in the base tool are actually plug-ins (such as the import/merge feature). While the API provides wide exposure to the underlying architecture and offers developers the hooks to build advanced features, the documentation for the API is minimal. In this paper we will explore the step-by-step process of building a plug-in, the GUI hooks that are easily accessible through the DI Studio plug-in API for surfacing plug-ins, and some of the various design approaches that should be considered when designing a plug-in.

Of the out-of-the-box features delivered with DI Studio, AutoMap is perhaps one of the most helpful (as well as problematic). AutoMap is a feature that attempts to determine the columns that should be mapped between individual job nodes, based on the column name, type, and length, and “maps” these columns without requiring the user to do so manually. AutoMap is triggered by the system rather than by the user. This feature, therefore, has the potential to over-write or delete any mappings the user has specifically created in the event that changes are made to the job after the initial AutoMap process has taken place. Out-of-the-box, DI Studio allows users to control AutoMap settings at the job node level by right-clicking on job nodes in the process editor and checking or un-checking the AutoMap option.

The plug-in examined in this paper was designed to provide a single-click feature for turning AutoMap off completely at the job level. Rather than requiring the user to click on each individual job node, which could be very time consuming and error prone (especially in large jobs), this plug-in allows the user to right-click on a job and tell DI Studio to turn AutoMap Off for all nodes in the job.

AUTOMAP – FRIEND OR FOE

Before we explore the plug-in API, let’s examine the AutoMap feature to appreciate the reasons this plug-in is a valuable addition to the DI Studio feature suite. When building a job, developers will often start with a blank canvas. Developers will generally know the structure of the target table(s) they are attempting to load, but may have to make a number of transformations to the source tables to get the data into the desired target structure. These transformations may be trivial changes - such as sorting the data by a particular column, or appending rows from two tables with a shared structure and common column names into a single table. Often times, the transformations are more complex and require numerous interim changes to achieve the target table structure.

By default, AutoMap is turned on for new job nodes. This means that when a node is connected to the source and targets, DI Studio will query the SAS® Metadata Server to determine if it is possible to automatically build the column level mappings between the source and target tables. To build these mappings, DI Studio looks for columns that have the same name, type, and length. Columns meeting these criteria are automatically mapped to one another, saving the user time because these obvious mappings are taken care of behind the scenes.

But the "behind the scenes" part is where things get interesting. Some features in DI Studio are triggered by the user, such as submitting a job or adding a new node. Other features, such as AutoMap, are triggered by system events. For example, assume a large job has been built and a number of changes have been made beyond the generic auto-created column mappings. Changes to source tables or job nodes early in the job flow will cause DI Studio to trigger AutoMap and cascade any changes throughout the job flow – all without user notification! This means that there is a strong possibility that the tool is going to destroy your work if you are not careful to turn off AutoMap for every node in the job. It is easy to see how in large jobs this can be particularly painful. Saving early and often is a best practice to avoid this issue, but a better solution is to work on the job to a point where you are comfortable that AutoMap is no longer adding any value and then turn off AutoMap for all nodes in the job, ensuring that DI Studio will not make any unwanted changes.

At the SAS® Metadata Server level, jobs are represented as objects. These objects have properties representing various attributes (such as a collection of transformation activities representing each individual job node). At the individual job node level, there are metadata attributes that tell the system whether AutoMap is enabled or disabled for that specific node. By leveraging the DI Studio plug-in API, one can programmatically manipulate the metadata to put a job into a desired state. To disable AutoMap, one needs only trace through all the nodes stored in metadata and either set (or unset) particular properties on the job nodes to tell the SAS® Metadata Server that AutoMap is disabled for all nodes. Out of the box, this feature is surfaced by right-clicking on a node in the process editor at the job node level. But in a large job, turning off AutoMap manually on all job nodes is both time consuming and error prone. Missing a single node leaves the job in an undesirable state when there is a possibility that DI Studio triggers AutoMap and causes changes to be overwritten. Wouldn't it be nice if you could click on a job and turn off AutoMap at the job level just as easily as you can delete or copy a job?

INTRODUCTION TO DI STUDIO PLUG-INS

DI studio is built using a commonly understood design pattern for custom plug-ins. This means that a developer can write some custom code using the SAS DI Studio API and then drop that code into the plug-ins directory and the system will pick up that new plug-in and load it the next time DI Studio starts.

The default DI Studio installation directory structure on windows is as follows:

```
C:\Program Files\SAS\SASETLStudio\9.1
C:\Program Files\SAS\SASETLStudio\9.1\docs
C:\Program Files\SAS\SASETLStudio\9.1\plugins
C:\Program Files\SAS\SASETLStudio\9.1\roadmaps
C:\Program Files\SAS\SASETLStudio\9.1\security
```

Plug-ins are packaged as Java jars and placed in the "plugins" directory. The manifest in the jar file provides details about how the plug-in is to be run, indicating details such as which class DI Studio should invoke when attempting to load the plug-in. The root class of the plug-in will extend a particular DI Studio class or implement a particular interface to determine how the plug-in appears to the user. Plug-in developers have wide latitude for how they go about implementing plug-ins so long as they adhere to this specification.

While an exhaustive investigation of the plug-in API and DI Studio source classes is beyond the scope of this document, let's examine the API briefly to better understand what it enables a developer to accomplish.

GUI HOOKS

Any plug-in will need to be surfaced to the user somewhere within the DI Studio client. There are a number of possibilities, such presenting a new option when a user right clicks on an object or when a user selects a particular wizard such as the "New Object" wizard. Other possibilities include adding a new tab to an existing dialog for additional features, such as adding a "Changes History" tab to the Table Properties dialog. Perhaps the objective of the plug-in is to help with management of code – the API supports hooks for adding new folders to the custom tab of the tree browser. After identifying the appropriate place to surface the feature, the developer will need to identify the appropriate interface to implement or class to extend when designing the root class of the plug-in.

Here are some of the interfaces in the DI Studio Plug-In API for surfacing new plug-ins to the user. These are all located in the sas.framework.workspace.jar file under the package name com.sas.workspace.plugins (with the

exception of PluginInterface contained in sas.framework.plugins.jar and the package com.sas.plugins). Note, a plugin may implement as many of these interfaces as they want (or as needed):

Interface	Use
PluginInterface	The main plugin interface. All plugins must implement the methods contained in this interface.
AboutInterface	Display copyright, and general information about a plugin
MainTabbedPanePlugin	To add components to a tabbed dialog
MetadataExporterInterface	Plugins that read metadata from the repository and export it (the standard DI Studio export tool is written as a plugin)
MetadataImporterInterface	Plugins that import metadata and write to the repository (the standard DI Studio import tool is written as a plugin)
MetadataVisualsPluginInterface	Specify Icons and menu items for a plugin
MultipleNodeInterface	Allows for multiple sets of information to be exported from the plugin (multiple names, descriptions, icons, etc…)
PluginViewInterface	For assigning property tabs
TreeNodeInterface	For adding new types of tree nodes to existing trees
WorkspaceTreeContextMenuPluginInterface	Add entries to the drop down menu in the workspace trees of the project, custom, and inventory panels

SWING-ISH COMPONENTS API

Java developers who have done Swing coding in the past or developers who have used other plug-in architectures will be familiar with a number of common paradigms for building GUIs. For example, the layout of parts of a window and the type of window are foundational parts of building any plug-in that will present itself as a pop-up. Right-mouse-button click menus are triggered by capturing different events than are features triggered by left-mouse-button clicks. Other common design patterns are modal dialog windows and wizards, which give the developer some control over the flow through a series of steps in a plug-in.

Although past experience with Swing coding or plug-in development will be helpful, there is a learning curve when building DI Studio plug-ins because the API is unique to SAS and many of the traditional GUI components are custom SAS implementations that provide metadata lifecycle hooks. This means that it is possible to leverage built-in features of the API to save data to the metadata server when a window is closed, or to ignore changes. It also means it is possible to get a reference to a connection to the metadata server to query for information about a particular object.

ACCESSING METADATA

The SAS® Metadata Server is a remote storage component. In this client-server environment, the creation and persistence of objects is critical for developers to understand and when designing plug-ins. Objects retrieved from the metadata server must be correctly retrieved using APIs that return fully constituted objects else they will not be able to save any changes made. Changes made to objects must be correctly discarded to avoid unintended updates to metadata. Object stores should be instantiated and disposed of appropriately.

Of special note, the DI Studio client caches metadata from the server. This done for a variety of reasons (performance being the main one) but what it means for the developer is that changes made to the metadata must be flushed back to the server before they are permanently saved.

Metadata can be pulled/stored via a set of API's available off of the MdObjectFactory class. This factory class provides a series of methods for reading and writing metadata. It also provides for lifecycle methods for managing local copies of the metadata. In general, you must adhere to the following steps for the proper management of data from the server:

1) create an Object Store. An object store can be thought of as a local cache for short term management of metadata objects.
2) Create or pull metadata into the store.
3) Make local changes to the metadata
4) Write or flush the changes to the backend repository
5) Free the Object store

As a simple example, here is some sample code that creates a local object store and the reads metadata from the server:

```
// create an object store
MdObjectStore mdsAction = MdObjectFactory.createObjectStore(null, "Toggle");

// instantiate a job from a metadata stub
Job j = (Job) MdObjectFactory.createComplexMetadataObject(
    mdsAction,
    nodeToProcess.getMetadata()
);

… make some changes …

// write the changes back to the repository
j.updateMetadataAll();

// cleanup the object store
mdsAction.dispose();
```

You may notice that the createComplexMetadataObject takes an object as a parameter that is returned from getMetaData(). DI Studio has multiple representations of metadata – the simple, light version, and the heavyweight, full instance version. The tree node entries that we will modify use the lightweight version of the metadata. In order to get the full instance for processing we must pass the metadata stub to createComplexMetadataObject. This is an important concept. If you are only using the metadata stub, most of the values that you want to manipulate will not be there when you go looking for them!

THE WORKSPACE

There are a number of useful objects and concepts that are surfaced in the API that you will need to know about. When your code is invoked, it is invoked in the context of a Workspace. You can think of a Workspace as a set of global values and methods that you can use to query the current state of the DI Studio client. The most basic form is the Workspace.getWorkspace() call. This call returns an instance of the Workspace and can be used to query things like whether or not you have a project repository under change management.

The Workspace is also the top level component in the GUI. The workspace extends the JFrame class and represents application from a Swing standpoint. You can use this class to center dialogs, parent/register other GUI components, and to return the general state of the GUI.

A DI STUDIO PLUG-IN TO TURN OFF AUTOMAP

To turn off AutoMap on all nodes in a Job we must first understand what aspect of the metadata we are modifying. In the case of AutoMap, it turns out that the option is controlled by an Extension in a Property Set, as well as ***lack*** of the Extension. In a brand new job, before any node has had AutoMap turned off, the lack of the Extension indicates that the option is ON (don't ask how the authors feel about this design decision). What is an Extension? An Extension is a piece of metadata that hangs off another piece of metadata. In our case it hangs off of the PropertySets element that is attached to each node in the job. Using the "metabrowse" command from the DMS command box, we can drill down into a Job to discover whether or not the AutoMap option has been set.

The metadata hierarchy path that must be investigated is the following: <Job Name>, JobActivities, New Transformation Activity, Steps, <Step Name>, PropertySets, AUTOMAP, Extensions, AUTOMAP. If the Step node that you are investigating has not toggled the AUTOMAP option at least once then you will not see anything below PropertySets.

Figure 1 and Figure 2 show what Metabrowse look like for both instances. In all cases, to write a tool to turn off AutoMap you must handle both the case where there is no existing Extension for the Step's PropertySets, and the case where there is an Extension already. In fact, you must also handle the case where the AUTOMAP PropertySets exists ***without*** an AUTOMAP Extension. Older jobs appear to have had a different standard for specify the AutoMap option. You should plan to handle all cases.

Figure 1 - AUTOMAP has not been toggled yet. AutoMap is defaulted to ON in this case.

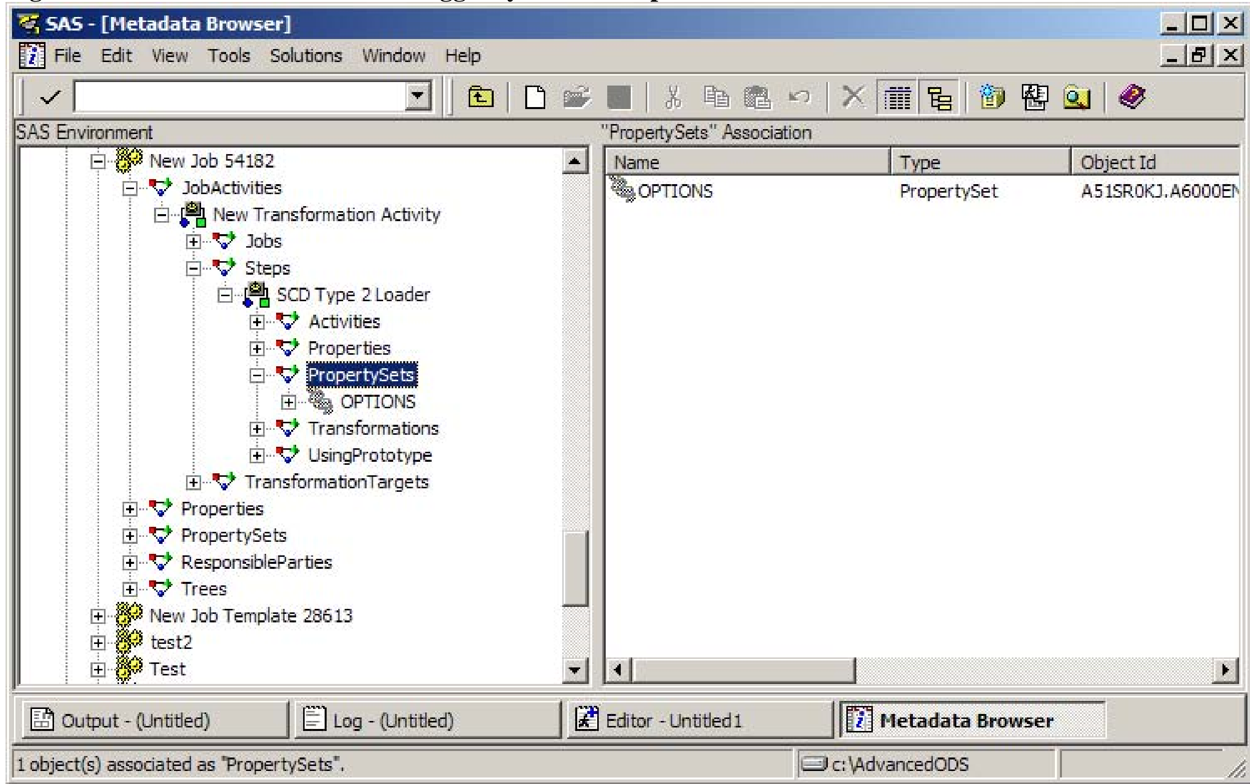
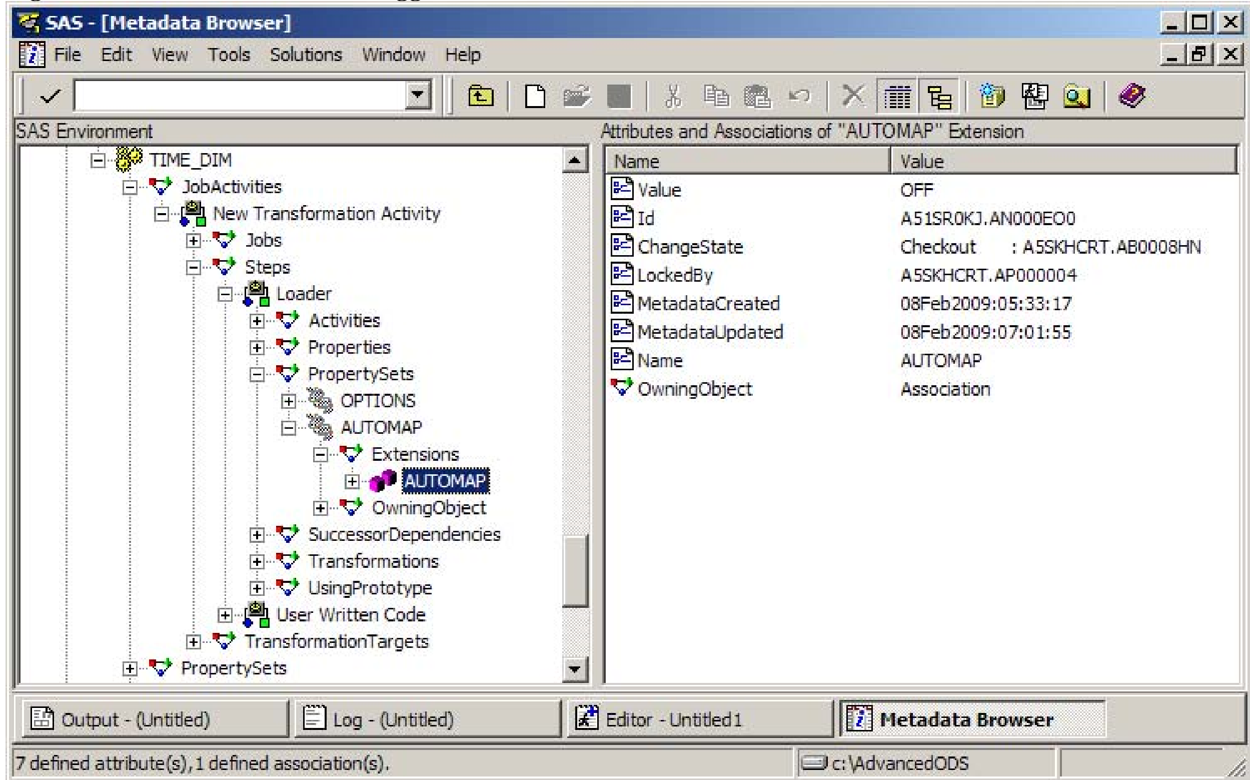


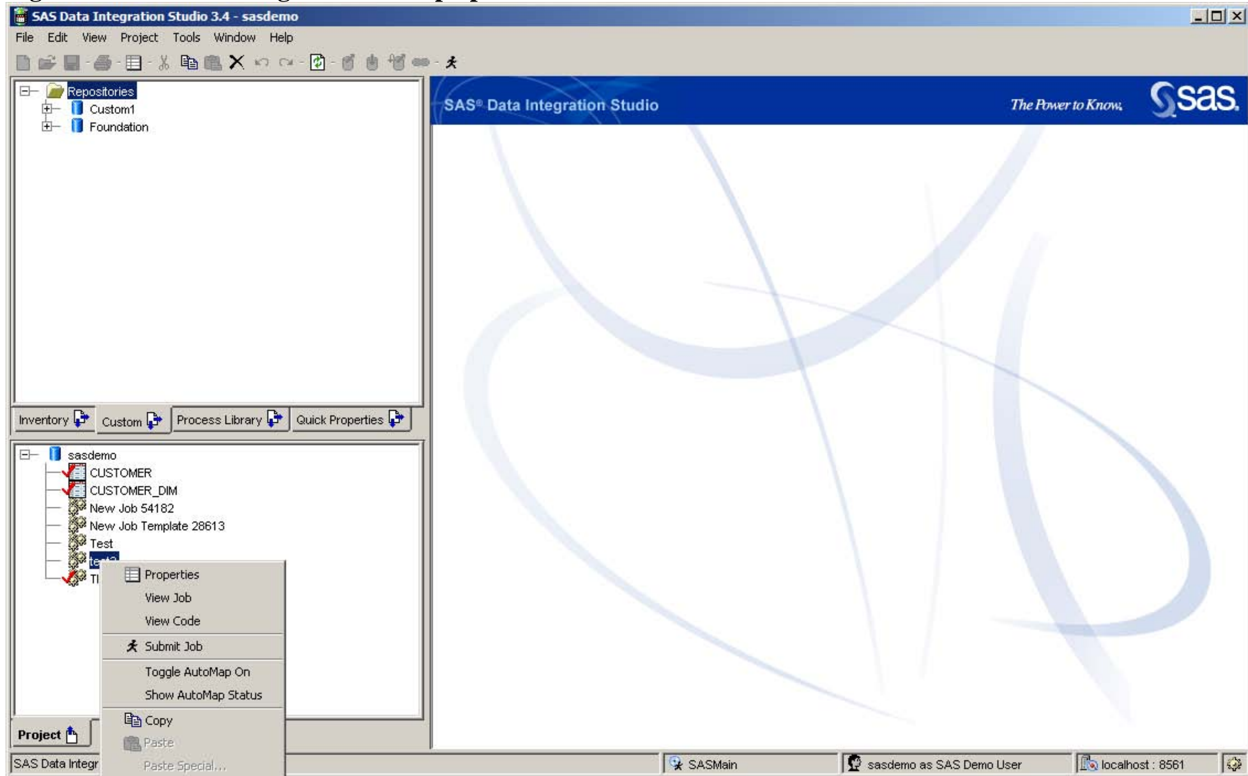
Figure 2 - AUTOMAP has been toggled at least once. Value has been set to OFF.



THE GUI

For this exercise, we are going add two options to the dropdown menu of a Job. This menu can be invoked by right clicking on a Job in one of the trees in the Project, Custom, or Inventory tabs. Figure 3 shows what the Menu items will look like:

Figure 3 - Menu showing the AutoMap options



We are going to add two menu items – one to toggle AutoMap for each node in the Job, and one to show the status of each node in the Job. The Status option will allow us to easily see if there are any nodes that AutoMap turned on. This is important as it may be beneficial to keep AutoMap on for certain nodes. We need to be able to investigate whether or not a node has AutoMap turned on without having to open every single node in the Job.

Figure 4 - Automap Status Dialog.

AutoMap Status

Node Name	AutoMap Status
Loader	On
Fact Table Lookup	On
SQL Join	On
SQL Join	On
SQL Join	Off

To create a plug-in that is accessible from the tree menu you must implement the WorkspaceTreeContextMenuPluginInterface interface. This interface is very simple – it only requires that you implement one method:

```
public JMenuItem[] getMenuItems(TreeNode[] nodes);
```

This method should return an array of Swing JMenuItem objects that represent the menu items to insert into the menu. The list of tree nodes that were selected is passed to you via the nodes array. Our algorithm then is simple:

1) inspect the nodes passed to us to see if they are Jobs
2) if not, ignore
3) if so, then create our two menu items. Each menu item must declare the action to take when the item is selected in the menu. We will code these actions in the callback method addActionListener()
4) when the menu item is selected the action is invoked. Each action should read and update any metadata as necessary.

To inspect the metadata to see if the item selected is a Job we must first instantiate the metadata. This can be achieved by calling createComplexMetaDataObject(). We can then surf the metadata tree to see if any nodes have the AutoMap bit set:

```
// did they pass us invalid tree nodes?
if (nodes == null || nodes.length != 1 || nodes[0] == null)
   return null;

// cast to the correct type
WsTreeNode wNode = (WsTreeNode) nodes[0];

// get the metadata stub for the object and check that it is a Job
CMetadata cNode = (CMetadata) wNode.getMetadata();
if (cNode == null || !cNode.getCMetadataType().equals("Job"))
   return null;

// Save the node that we are processing
final WsTreeNode nodeToProcess = wNode;

// Create an Object Store to hold a local copy of the metadata
MdObjectStore mdsAction = MdObjectFactory.createObjectStore(null, "Toggle");

// Create the full copy of the metadata including all options
Job j =
   (Job) MdObjectFactory.createComplexMetadataObject(mdsAction, nodeToProcess
     .getMetadata());
if ( j == null ) return null;

// surf the metadata and return a list of node info objects that tell us the name
// and whether or not the AutoMap bit is set
List jnodes = null;
try {
   jnodes = AutoMapToggler.getListOfNodeNamesAndAutoMapStatus(j, mdsAction);
}
catch (MdException mde) {
   jnodes = null;
   mde.printStackTrace();
}

// if any node in the Job has AutoMap set then set a flag so we can insert the correct
// type of menu entry
boolean automapOn = false;
for (int i = 0; i < jnodes.size(); i++) {
   JobNode jobNode = (JobNode) jnodes.get(i);
   if (jobNode.isAutomapEnabled()) {
      automapOn = true;
      break;
   }
}
```

THE BACKEND

The getListOfNodeNamesAndAutoMapStatus(j, mdsAction) and jobNode.isAutoMapEnabled() calls are where we investigate the metadata that was retrieved from the server. The first call traverses the metadata tree and checks to see if AutoMap is enabled. If so, it sets a boolean in the jobNode:

```
// check each Step in the Job to see if automap is enabled. Return a list of jobnodes
public static List getListOfNodeNamesAndAutoMapStatus(Job job, MdObjectStore oStore)
  throws MdException {
  List l = new ArrayList();

  // get the activities list from the Job and loop over them
  AssociationList jobActivities = job.getJobActivities();
  for (Iterator outer = jobActivities.iterator(); outer.hasNext();) {

     // get the transformation activity Steps and loop over them
     TransformationActivity ta = (TransformationActivity) outer.next();
     AssociationList steps = ta.getSteps();
     for (Iterator inner = steps.iterator(); inner.hasNext();) {
        TransformationStep currentTransformationStep = (TransformationStep) inner.next();

        // add the step to a new JobNode and check to see if autoMap is enabled
        l.add(new JobNode(currentTransformationStep.getName(),
           isAutoMapEnabled(currentTransformationStep)));
     }
  }

  return l;
}

// check the Transformation Step to see if AutoMap is enabled
private static boolean isAutoMapEnabled(TransformationStep ts) throws MdException {

  // get the property set from the Step and loop over them
  AssociationList propertySets = ts.getPropertySets();
  for (Iterator iterator = propertySets.iterator(); iterator.hasNext();) {
     PropertySet ps = (PropertySet) iterator.next();

     // get the Extensions from the property set and loop over them
     AssociationList extensions = ps.getExtensions();
     for (Iterator iterator2 = extensions.iterator(); iterator2.hasNext();) {

        // Check the AutoMap extension
        Extension ext = (Extension) iterator2.next();
        if ("AUTOMAP".equalsIgnoreCase(ext.getName())) {

           // value is set to OFF. Return false.
           if ("OFF".equalsIgnoreCase(ext.getValue())) {
              System.out.println("  AUTOMAP: This step has automap 'OFF'");
              return false;
           }

           // value is set to ON. Return true.
           else if ("ON".equalsIgnoreCase(ext.getValue())) {
              System.out.println("  AUTOMAP: This step has automap 'ON'");
              return true;
           }
        }
     }
  }

  // this is important! If we can’t find the AutoMap Extension then the AutoMap
  // is ON by default! Return true.
  return true;
}
```

If it is determined that any nodes in the Job have AutoMap enabled then we will add a menu option to “Toggle AutoMap Off”. If no nodes in the Job have AutoMap enabled then we will add an option to “Toggle AutoMap On”. Note, if the option is selected we modify the state of the Job and the Job is saved back to the repository. You can do this even while the Job is open in the process flow diagram (PFD) editor. If any other changes are currently active in the Job they will not be saved until you close and save the PFD.

How is metadata added via the Java API? To add the option to turn on AutoMap we must write metadata back to the server. This is accomplished by creating metadata and the calling updateMetadataAll() on the Job:

```
// method to set the AUTOMAP extension bit
private static void turnOnAutoMapping(TransformationStep ts, String jobReposId,
   MdObjectStore oStore) throws MdException {


   // surf the property sets as before
   AssociationList propertySets = ts.getPropertySets()
   for (Iterator iterator = propertySets.iterator(); iterator.hasNext();) {
      PropertySet ps = (PropertySet) iterator.next();

      // get the extensions and surf
      AssociationList extensions = ps.getExtensions();
      for (Iterator iterator2 = extensions.iterator(); iterator2.hasNext();) {
        Extension ext = (Extension) iterator2.next();

        // if we hit the AUTOMAP extension set the value to turn it
        if ("AUTOMAP".equalsIgnoreCase(ext.getName())) {
          if ("OFF".equalsIgnoreCase(ext.getValue())) {
            System.out.println("  AUTOMAP: This step has automap 'OFF' - setting it to ON.");
            ext.setValue("ON");
            return;
          }
          else if ("ON".equalsIgnoreCase(ext.getValue())) {
            System.out.println("  AUTOMAP: This step already has automap 'ON'");
            return;
          }
        }
      }
   }
}
```

To turn AutoMap off we have to check two cases. If the AutoMap option already exists then we can just update the value. If the Extension does not exist then we must add it:

```
private static void addAttributeForAutoMapOff(TransformationStep ts, String jobReposId,
  MdObjectStore oStore) throws MdException {

  System.out.println("  AUTOMAP: No automap reference - adding one, setting it to OFF.");

  // does auto map property set already exist? Find the property set named AUTOMAP
  AssociationList propertySets = ts.getPropertySets();
  PropertySet _propSet = null;
  for (Iterator iterator = propertySets.iterator(); iterator.hasNext();) {
     PropertySet ps = (PropertySet) iterator.next();
     if ( ps.getName().equalsIgnoreCase("AUTOMAP") ) {
        _propSet = ps;
        break;
     }
  }

  // create a propertySet if it does not exist already.
  if ( _propSet == null ) {

     // allocate the object
     _propSet = (PropertySet) MdObjectFactory.createComplexMetadataObject(oStore, null, "",
          MdObjectFactory.PROPERTYSET, getReposIdForRepository(jobReposId));

     // set it's name to AUTOMAP
     _propSet.setName("AUTOMAP");

     // set the owner (parent) to the Step that we are processing
     _propSet.setOwningObject(ts);
  }

  // create an extension for the AUTOMAP option and set the value
  Extension _ext =
    (Extension) MdObjectFactory.createComplexMetadataObject(oStore, null, "",
        MdObjectFactory.EXTENSION, getReposIdForRepository(jobReposId));
  _ext.setName("AUTOMAP");
```

```
      _ext.setValue("off");

      // add the extension to the propertySet
      _propSet.getExtensions().add(_ext);
    }
```

Each of these methods can be called from our menu item. The menu item to toggle AutoMap calls the following action when the user selects it:

```
   // perform the update action when the user selects it
   public void actionPerformed(ActionEvent e) {

      // create a local object store to handle any changes
      MdObjectStore mdsAction = MdObjectFactory.createObjectStore(null, "Toggle");
      Job j = (Job) MdObjectFactory.createComplexMetadataObject(mdsAction,
                       nodeToProcess.getMetadata());

      // toggle the value of the AutoMap option
      if (j != null) {
        try {
          if ( autoMapIsOn )
            AutoMapToggler.turnAutoMapOffForJob(j, mdsAction);
          else
            AutoMapToggler.turnAutoMapOnForJob(j, mdsAction);
        }
        catch (MdException e1) {
          e1.printStackTrace();
        }
      }

      // update the metadata! Without this call then changes are not saved.
      try {
        j.updateMetadataAll();
      }
      catch (MdException e1) {
        e1.printStackTrace();
      }

      // cleanup after yourself
      mdsAction.dispose();
   }
```

For SAS programmers, the above Java code may seem cryptic. However, you can make similar changes to the metadata using PROC METADATA. The technique for doing this is beyond the scope of this paper, and you should note that using the PROC will have to be run from a job or stand alone SAS program. It is entirely conceivable that you could run a post process job to update all of the AutoMap options for each Job in the repository!

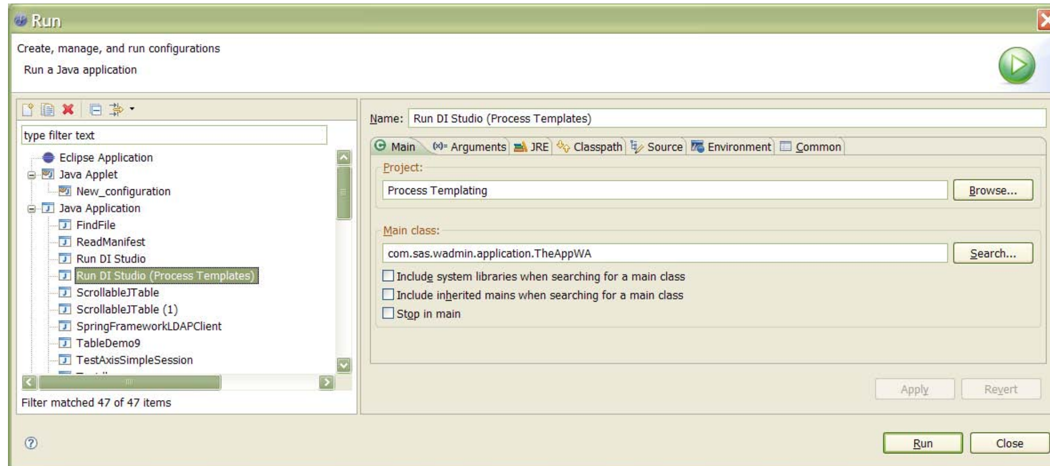
WORDS OF WISDOM FROM DI STUDIO PLUG-IN DEVELOPERS

One of the great things about java code is that it can be decompiled to allow developers to trace through the call stack. Our team developed the DI Studio AutoMap plug-in using the MyEclipse IDE. Assuming some basic conceptual understanding of debugging java code, the biggest take-away from this section should be an appreciation that when developing DI Studio plug-ins, the Eclipse debugger is your friend. If you couple the Eclipse debugger with a de-compiler then you have a complete IDE for understanding the call stack and object lifecycles within the DI product. The most painful part of developing DI Studio plug-ins (short of the lack of documentation) is the fact that you have limited ability to test your code without actually packaging the plug-in and launching it within DI Studio. One of the most enabling tips we can offer is to configure Eclipse to launch and debug DI Studio so you can run plug-ins and step through the call stack via the debugger.

Here's how we launch DI Studio via Eclipse. The reason for doing so is to set breakpoints in the java source and have DI Studio pause so you can step through your code. That being the case, this assumes you have a project in Eclipse with all the necessary jar files (your source and some of the DI source).

1. Create a run configuration. Name it whatever you want, associate it with the project that has the DI studio and source files you want to debug.

2. On the "Main" tab, put "com.sas.wadmin.application.TheAppWA" in the main class. This class is in sas.dbuilder.app.jar in the root DI Studio install directory.



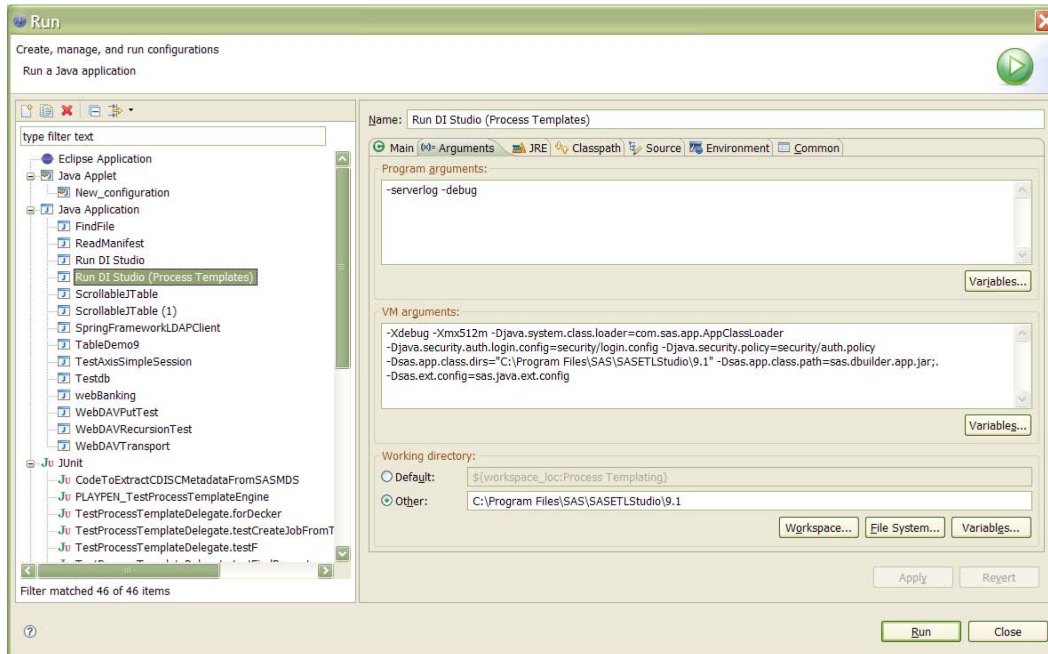
3. On the arguments tab, you have to add some things to enable debugging.

   a. In "Program Arguments" add **-serverlog -debug**

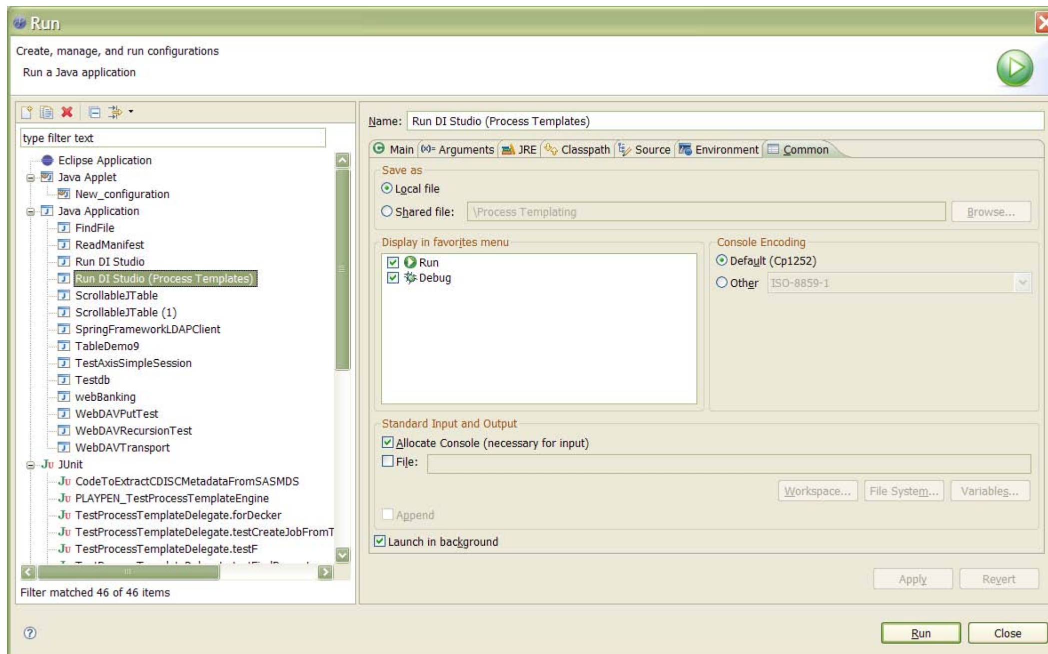
   b. In "VM Arguments" add **-Xdebug -Xmx512m -Djava.system.class.loader=com.sas.app.AppClassLoader -Djava.security.auth.login.config=security/login.config -Djava.security.policy=security/auth.policy -Dsas.app.class.dirs="C:\Program Files\SAS\SASETLStudio\9.1" -Dsas.app.class.path=sas.dbuilder.app.jar;. -Dsas.ext.config=sas.java.ext.config**

   c. Set the working directory to the DI Studio home directory.

4. Go to the common tab and enable the checkboxes to show this run configuration in Run and Debug menus



5. Now if you “run” this configuration in Eclipse, DI Studio should launch. If you “debug” this configuration in eclipse, you should be able to set breakpoints and step through the code. In the “console” view in Eclipse, you should see the DI Studio XML chatter back and forth from the server displayed.



6. Note: you can achieve this same debugging output by putting these same options in the etlstudio.ini file in the DI Studio root installation directory, but you will NOT be able to put breakpoints in your code. You can get breakpoints by adding some additional VM options in the etlstudio.ini file, but that route actually modifies the start-up configuration for DI Studio *in all cases*, rather than just when using eclipse. The side effect is that the overhead from running in debug mode is incurred always, rather than just when you launch DI Studio via Eclipse.

The steps discussed above will give you the ability to debug your project within DI Studio, but unless you have the DI Studio source code it is still challenging to understand the call stack and navigate the source code for the classes and interfaces of the plug-in architecture. Perhaps you can negotiate with SAS for a copy of the source code. But if you cannot or time doesn't give you that luxury, we point you at Jadclipse (http://jadclipse.sourceforge.net/).

Jadclipse is an open source Eclipse plug-in that takes the JAD de-compiler and integrates it into Eclipse so you can browse jars for which you don't have the source code. The jars for DI Studio are not all named intuitively, but you can access the class files fairly easily by including the jars from the DI Studio installation directories within your Eclipse project and configuring Jadclipse to decompile these classes. Although the decompiled sources will in many cases be less than ideal (e.g. variable names are not always decompiled, method signatures do not include useful names, line numbers may not line up perfectly) they are way better than nothing. The installation and configuration of Jadclipse is well documented on the website.

CONCLUSION

Using the DI Studio plug-in API it is possible to add extensions to DI studio to make your life much easier. Getting rid of AutoMap is a worthwhile endeavor as it teaches the basics of the plug-in API and exposes you to the internal metadata framework.

The examples described in this paper are for Version 9.1.3 of the SAS system. In 9.2 (Version 4.1 of DI Studio), the tool has options to control how AutoMap works. However, there will still be an API for plug-ins and extensions. The authors are sure that opportunities will exist to write extension to the tool to cleanup of fix less than ideal behavior!

ACKNOWLEDGMENTS

The authors would like to thank the developers at SAS (especially Nancy Rausch and Russ Robison) for their patience and willingness to answer our endless stream of questions. Y'all are the best!

CONTACT INFORMATION

Your comments and questions are valued and encouraged. d-Wise Technologies is a software consulting company focused on delivering data warehousing, software integration, and business intelligence solutions across industries. d-Wise primarily focuses on delivering SAS solutions but has experience in a wide range of technologies. The founders of d-Wise have a combined 26 years working at SAS Institute in the Research Development group developing SAS products.

d-Wise is a SAS Institute Alliance Partner and preferred partner for delivering solutions specifically in the Life Sciences Industry including SAS Clinical Data Integration and SAS Drug Development. d-Wise is also an Associate Member of CDISC.

Contact the authors at:

Name: Chris Olinger, Stephen Baker
Enterprise: d-Wise Technologies, Inc.
Address: 3115 Belspring Lane
City, State ZIP: Raleigh, NC 27612
Work Phone: 888.563.0931
Fax: 888.563.0931
E-mail: Sales/Info - info@d-wise.com
Chris Olinger - colinger@d-wise.com
Stephen Baker - sbaker@d-wise.com
Web: www.d-Wise.com

SAS and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.

Other brand and product names are trademarks of their respective companies.